Продолжаем тему множественного наследования. На прошлом занятии мы с вами объявили два базовых класса GeomSerialize и GeomBase, от которых формируется дочерний класс Line. Пока эти классы содержат совершенно разные методы, никаких проблем их совместного использования не возникает. Но, что если в этих классах прописать метод с одинаковой сигнатурой? Например:

```language-cpp
class GeomSerialize {
...
    size_t get_size_obj() const { return sizeof(*this); }
};

class GeomBase {
...
    size_t get_size_obj() const { return sizeof(*this); }
};
```

Пока нет вызова этого метода, программа работает, как и ранее. Но, если попытаться его вызвать через объект дочернего класса Line:

```language-cpp
int main()
{
    Line ln(1, 2, 10, 20);

    size_t sz = ln.get_size_obj(); // ошибка

    return 0;
}
```

то компилятор выдаст ошибку, что метод get_size_obj неоднозначен, то есть, объявлен в обоих базовых классах и какой именно вызывать непонятно. О**дин из вариантов решения этой неопределенности – явно прописать область видимости одного из базовых классов, где этот метод прописан**:

```language-cpp
size_t sz = ln.GeomBase::get_size_obj();
```

Или же получить ссылку или указатель на один из классов и через него вызвать такой метод:

```language-cpp
GeomSerialize& lnk_sz = ln;
```

```
sz = lnk_sz.get_size_obj();
```

Но, во-первых, это выглядит несколько коряво, и, во-вторых, скорее всего не совсем то, что нам нужно. Часто подобная коллизия методов – это признак неверно описанных классов. Возможно, достаточно сменить названия методов и проблема будет решена. Однако тот же самый эффект может возникнуть даже если в классах не прописывать одинаковые методы. Они могут унаследоваться от вышестоящего класса.

Ромбовидное наследование

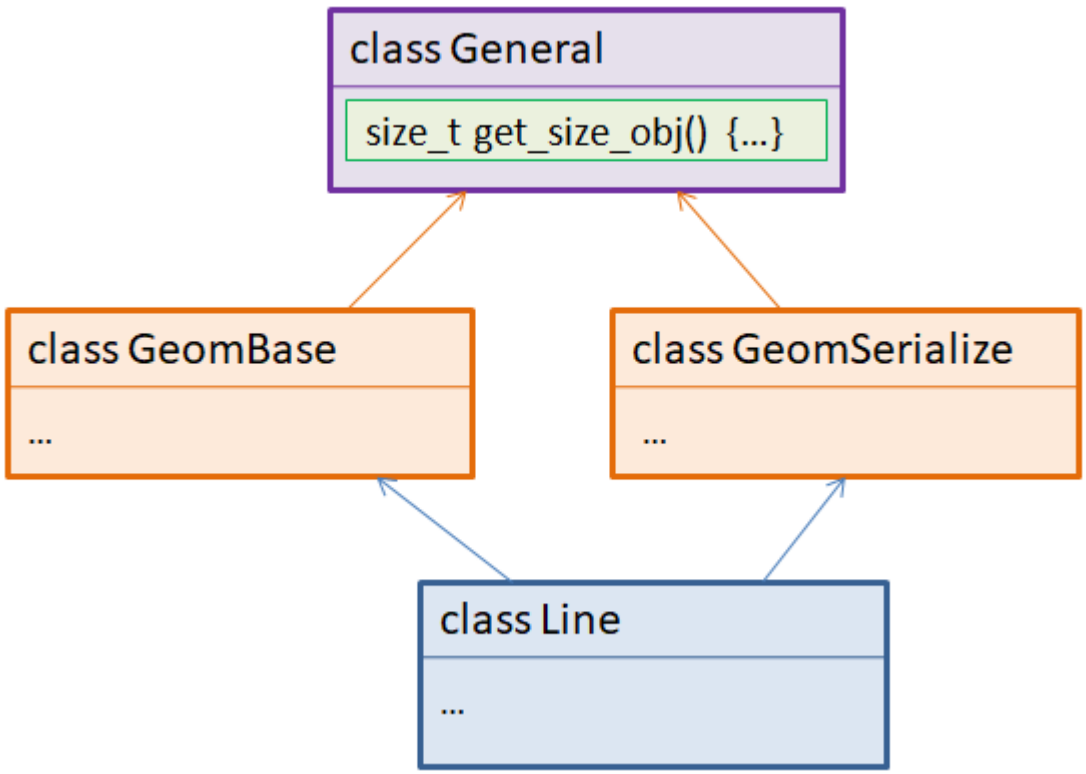
Например, перенесем в следующий по иерархии базовый класс General метод get_size_obj:

```language-cpp
class General {
public:
    size_t get_size_obj() const { return sizeof(*this); }
};
```

И унаследуем от него классы GeomSerialize и GeomBase:

```language-cpp
class GeomSerialize : public General { ... };
class GeomBase : public General { ... };
```

Получим следующую иерархию наследования:



В результате метод get_size_obj оказывается в области видимости обоих классов GeomSerialize и GeomBase. Соответственно, при его вызове через объект класса Line, возникает все та же неопределенность. Мало того, конструктор класса General будет вызван дважды. Если его явно прописать в классе General:

```language-cpp
class General {
public:
    General()
        { std::cout << "General: constructor" << std::endl; }

    size_t get_size_obj() const { return sizeof(*this); }
};
```

то после запуска программы увидим (при создании объекта класса Line):

```
General: constructor
GeomBase: constructor
General: constructor
GeomSerialize: constructor
Line: constructor
Line: destructor
GeomSerialize: destructor
GeomBase: destructor
```

Виртуальное наследование

Конечно, подобных ситуаций лучше избегать еще в процессе проектирования архитектуры программы. Если же использование такого наследования считается необходимым и оправданным, то спрашивается, как исправить возникшую ситуацию? Здесь нам на помощь приходит механизм **виртуального наследования** базовых классов.

Если при определении наследования классов GeomSerialize и GeomBase прописать ключевое слово virtual:

```language-cpp
class GeomSerialize : virtual public General { ... };
class GeomBase : virtual public General { ... };
```

То мы видим, как радикально меняется ситуация. Теперь объект класса General формируется только один раз и в единственном экземпляре содержится в объекте дочернего класса Line. Это как раз то, что нам

нужно. Благодаря этому метод get_size_obj так же появляется в единственном экземпляре. Поэтому вызов:

```language-cpp
size_t sz = ln.get_size_obj();
```

больше не приводит к каким-либо ошибкам на этапе компиляции программы. Причем, мы по прежнему можем вызывать этот метод с указанием той или иной области видимости:

```language-cpp
size_t sz = ln.GeomBase::get_size_obj();
```

или

```language-cpp
size_t sz = ln.GeomSerialize::get_size_obj();
```

Это будет все тот же самый один метод, прописанный в классе General. Дублирование не возникает благодаря использованию виртуального наследования.

Защищенный конструктор

Давайте еще раз посмотрим на наши полученные классы. Вероятно, что объекты классов General, GeomSerialize, GeomBase создавать в программе вряд ли необходимо. И было бы правильно явно запретить эту операцию. Конечно объекты абстрактных классов GeomSerialize и GeomBase мы так и так создать не сможем, но если чисто виртуальные методы будут заменены на обычные виртуальные, то ситуация изменится. **Один из надежных способов запрета создания объектов классов – это поместить их конструкторы в секции protected**. Почему именно protected, а не private? Если конструкторы пометить, как private, то нельзя будет создавать и объекты производных от них классов. Нам же нужно, только определить запрет создания объектов базовых классов. Поэтому выбираем секцию protected.

После изменения, запускаем программу, видим, что все работает по прежнему. Но, если попытаться создать какой-либо объект любого базового класса:

```language-cpp
int main()
{
    Line ln(1, 2, 10, 20); // ok
```

```
    General gb; // ошибка
    GeomSerialize* ptr_sz = new GeomSerialize; // ошибка

    return 0;
}
```

то возникнет ошибка на этапе компиляции программы.

Защищенный деструктор

А какой эффект можно получить, если в раздел protected или private поместить деструктор класса? Давайте это пропишем у класса Line:

```language-cpp
class Line : public GeomBase, public GeomSerialize {
    double length{0.0};
protected:
    ~Line()
        { std::cout << "Line: destructor" << std::endl; }
public:
...
};
```

Тогда мы сможем сформировать объект класса только с использованием оператора new следующим образом:

```language-cpp
int main()
{
    Line ln(1, 2, 10, 20);  // ошибка
    Line* ptr_ln = new Line(1, 2, 10, 20);  // ok

    delete ptr_ln;  // ошибка

    return 0;
}
```

Почему возникает ошибка в строчке:

```language-cpp
Line ln(1, 2, 10, 20);
```

Очевидно, здесь компилятор не может вставить вызов деструктора этого объекта, т.к. он является защищенным или приватным. Но создать объект с помощью оператора new все еще можно, т.к. на этом этапе вызов деструктора не предусмотрен. Он должен сработать в момент освобождения памяти в команде:

```cpp
delete ptr_ln;
```

Именно поэтому она также приведет к ошибке на этапе компиляции.

Таким образом, защищенный деструктор не позволяет формировать объекты класса в стековом фрейме, но разрешает в куче. Но, как же тогда удалять такие объекты и освобождать память? Для этого в классе Line достаточно прописать статический метод, например:

```cpp
class Line : public GeomBase, public GeomSerialize {
...
    static void delete_object(Line* ptr)
    {
        delete ptr;
    }
};
```

И вызывать его для удаления объектов этого класса:

```cpp
int main()
{
    Line* ptr_ln = new Line(1, 2, 10, 20);
    Line::delete_object(ptr_ln);
    return 0;
}
```

Этот прием с закрытым деструктором иногда полезно использовать на практике, если предполагается размещать объекты класса исключительно в куче.